Profile Generation and Guided Optimization for Compiled Queries

Final Report
May 5, 2021

Abi Kim, Kyle Dotterrer,
Wan Shen Lim
Carnegie Mellon University

1. INTRODUCTION

This document contains our final report for our final project in CMU's 15-745: *Optimizing Compilers* course.

1.1 Problem Statement

The goal of our project is to improve the runtime performance of query execution in a data-centric code generation DBMS engine by integrating dynamic optimization techniques. Specifically, we aim to implement profile-guided optimization for JIT-compiled queries in the NoisePage [4] LLVM-based execution engine. We partition this problem into the following sub-problems:

- Implement support for collection of low-level statistics and profile information in the NoisePage JIT. These statistics may include runtime information for individual operators at arbitrary abstraction levels that appear during the lowering process within the NoisePage execution engine. Examples of supported operators might include SQL operators, operator pipelines, and TPL functions. Adding support for collection of these statistics enables the implementation of SQL's **EXPLAIN ANALYZE** command. More importantly, the profile information collected with this mechanism may be used as input to other components of the system.
- Implement a feedback loop mechanism that allows for an automated feedback-driven exploration of the space of potential LLVM optimization passes, based on the profile information generated by prior iterations.
- Explore techniques for efficient instrumentation of fused operators. Sufficient gains in efficiency might allow us to gather and utilize profile information in an online setting, rather than merely across queries.

In addressing these problems, salient challenges arise from the following sources:

- **Query Compilation**: Collecting query execution profiles in database systems that execute queries in software can be accomplished via traditional software profiling techniques. These techniques break down for systems like NoisePage that compile queries to native code and execute them directly on hardware because the execution profile for the query is completely distinct from the execution profile for the system that compiled it.
- **Operator Fusion**: Execution profile collection is further complicated by the operator fusion optimization technique employed by the NoisePage execution engine [9]. In a less sophisticated execution engine, each operator in the physical plan tree for the query might be translated directly to a function in the code generated for the query. However, NoisePage employs the operator fusion technique in which multiple operators in the physical query plan may be fused into a single *pipeline* abstraction that is subsequently compiled to a single function in the generated code. While it improves query runtime by limiting temporary data materialization, operator fusion destroys the relationship between the generated code and the higher-level abstractions in the system.
- **The DBMS-Centric Context**: Some additional considerations introduced by the DBMS setting are:
  - Highly Multi-User: No single query should starve the entire system of resources.
  - Time to First Response Matters: It is not acceptable to spend too long optimizing instead of quickly responding. We must locate an appropriate tradeoff between optimization time and the corresponding runtime performance improvement.
  - Adaptive Execution Considerations: Fast OLTP queries are run through an interpreter, long OLAP queries are compiled in the background and swapped in when compilation is complete. This model lends itself well to multiple stages of compilation with different optimization levels. We are unaware of existing projects that draw a link between adaptive execution and profile-guided optimization.

1.2 Approach

We partition our approach in this project into the following two thrusts:

Thrust #1: Implement support for execution profile generation in the NoisePage execution engine. Near the midpoint of the project timeline, the TUM group published a paper at EuroSys 2021 that presents a general solution to the problem of profile generation across abstraction levels in dataflow systems [10]. The *tailored profiling* technique introduced in this work is highly applicable to the implementation of our profiling subsystem and provides a roadmap for successfully solving one of the primary problems we address in this project. We adopt the implementation of tailored profiling in our implementation of a profiling subsystem within NoisePage.

Thrust #2: Implement an automated feedback-driven exploration of the space of potential LLVM optimization passes, applied at whole-program or individual-function granularity.

1.3 Related Work

The recent EuroSys 2021 publication by Beishcl et al. is both timely and highly-relevant to the implementation of our profiling subsystem [10]. This paper describes a general solution to the problem of profiling dataflow systems at an abstraction level that is appropriate for the problem domain. Furthermore, this paper also describes integration of tailored profiling with the data-centric code generation engine within the Umbra [8] DBMS, making it a true model for our system. Prior works in this area present solutions that are applicable within limited domains [13] or require substantial manual effort during post-processing [11].

There are separate branches of work that relate to optimizing queries for a code generation DBMS engine. Examples of code generation DBMS engines include HyPer [1] and SingleStore [6], but typically all of the program's code is compiled at the same optimization level with the same flags. Projects that varied compilers and compiler flags to find the best possible code for a primitive have been explored in the context of the Vectorwise [12] system. However, the implementation in Vectorwise relies on a-priori knowledge of the primitives that will be compiled. This approach is therefore less flexible than the implementation within NoisePage which can generate arbitrary functions at runtime without any a-priori information.

1.4 Contributions

In this project, we make the following contributions:

- Explore the feasibility of implementing a profiling subsystem via tailored profiling [10] in the NoisePage DBMS. A key insight is that often, an optimizing compiler must make decisions based on limited or even no information as to how its optimizations will affect the overall programâĂŹs efficiency. By implementing a profiling subsystem that can track profile data across multiple abstraction levels, a profile-guided optimizer will have a much richer set of information to understand the budget, scope, and impact of any optimizations made.
- Explore the feasibility of automated search strategies for determining what set of LLVM transform passes to apply and at what granularity, e.g., whole-program vs function-level. We find that automated search is indeed possible to implement.
- Evaluate our strategy against a set of hand-picked globally-applied LLVM function transform passes that were empirically chosen by a DBMS+compilers domain expert Dr. Menon. While we find that Dr. Menon is very difficult to defeat on similar grounds (whole-program optimization), we find that our strategy is effective at rapidly sampling the space of potential optimizations to find promising and cheap transformations for a given SQL query, which allows for a number of exploratory techniques such as iterating on the domain expert's knowledge with genetic algorithms, completely random sampling of complete pipelines in hopes of lottery tickets, etc.
- In the NoisePage DBMS, and perhaps similar DBMSes which have integrated a separate query processing engine DSL by grafting new storage layer builtins into the language, it actually doesn't matter whether you apply LLVM transform passes or not, and in fact you save more microseconds by performing 0 transform passes overall.

The remainder of this document is structured as follows. First, in Section 2, we provide a detailed description of our design and approach. Next, in Sections 3 and 4, we describe our experimental setup and the results of our evaluation. Section 5 discusses some surprised and lessons learned over the course of the project. We conclude in Section 6 and comment on potential directions for future work.

2. DESIGN AND APPROACH

2.1 Profiling across Abstraction Boundaries

In order to provide the automated optimization exploration framework with richer information than just total execution time, we implement a profiling subsystem within NoisePage. This subsystem allows us to collect a fine-grained execution profile for JIT-compiled query code.

2.1.1 Implementation False Starts

We made two initial attempts at implementing the profiling subsystem that ultimately proved to be false starts:

- **Attempt #1: Solve it in C++** We attempted to modify the C++ translators that convert DBMS physical plans to `TPL` code by inserting calls to instrumentation functions. This proved tedious, unwieldy, error-prone, and provided only low-resolution information.
- **Attempt #2: Solve it in `TPL`** We have (and continue to use) implemented the recording of information at execution time into DSL-level structs. This makes it convenient to record features such as the number of entries into a given function. However, this requires all the information that you want to record to be exposed at the DSL level itself, which may not always be practical and introduces unnecessary complexity.

We abandoned both of these approaches upon learning of the general solution to profiling across abstraction levels presented by Beischl et al. [10].

2.1.2 Final Design Overview

The final design of our profiling subsystem is based on the tailored profiling design described by the Beischl et al. [10].

The profiling subsystem assumes responsibility for two distinct tasks:

- **Tagging Dictionary Construction**: The *tagging dictionary* is a multi-level dictionary data structure that allows us to effectively undo the lowering performed by the NoisePage execution engine during the code generation and query compilation process. Each level of the tagging dictionary maps individual operations in some intermediate representation to the operation at the next higher level of abstraction that produced it. The tagging dictionary is constructed as the query is lowered from the physical query plan tree to native machine code.
- **Runtime Sample Collection**: We utilize the Linux perf API [3] to collect an execution profile for JIT-compiled queries as they are executed.

2.1.3 Tagging Dictionary Construction

Our implementation of the tagging dictionary recognizes the following five intermediate representations (from highest to lowest abstraction level):

- SQL Operator
- `TPL` AST Node

- TPL Bytecode
- LLVM IR
- x86 Machine Code

We utilize a tagging dictionary with three levels to record the translations that are performed across the entirety of the lowering process. The lowest level mapping (from x86 to LLVM IR) is generated without manual population of the tagging dictionary by parsing the debugging information emitted by the LLVM JIT when the native code object is originally produced.

A brief description of the implementation required to populate each level of the tagging dictionary is provided below.

- **TPL AST Node to SQL Operator**: We populate this level of the tagging dictionary during the first phase of code generation. We modify the code generator's operator translators to emit tags whenever a new AST node is constructed.
- **TPL Bytecode to TPL AST Node**: We populate this level of the tagging dictionary during the second phase of code generation. We add functionality to the TPL compiler that allows us to inject a callback function that emits a tag whenever the bytecode generator emits a new bytecode instruction.
- **LLVM IR Instruction to TPL Bytecode**: We populate this level of the tagging dictionary during query compilation. We manually insert calls to emit tags whenever a new IR instruction is inserted into the compiled module.

2.1.4 *Profile Sample Collection*

We utilize the Linux perf API to collect an execution profile for the JIT-compiled query as it is executed. We utilize event-based sampling on instruction retirement (`INST_RETIRED.PREC_DIST`) events. A sample is collected every 5,000 events. To ensure the accuracy of the collected samples, we request precise instruction pointers (0 skid); this enables the Processor Event Based Sampling (PEBS) feature on Intel platforms [2]. This sampling configuration is the same as that utilized by Beischl et al. in their proof-of-concept [10].

Samples are written to a memory-mapped ring buffer by the kernel at the specified interval. We read the raw samples from this ring buffer and serialize them to a JSON file for subsequent processing.

2.1.5 *Tailored Profiling*

We combine the information collected in the tagging dictionary with the samples collected during query execution to generate profiles for any intermediate representation within NoisePage. For example, tailored profiling enables implementation of the SQL **EXPLAIN ANALYZE** command by mapping native machine instructions all the way back up to SQL operators. However, it might also be useful to analyze profiles for lower levels of abstraction such as TPL functions. Our implementation of tailored profiling is flexible enough to support arbitrary post-processing of this kind.

2.2 Programmatic Pass Application

We are unable to use LLVM's new non-legacy function pass manager for some annoying build-related reasons. By simply assigning each pass an ID, (in fact just the index of the pass in the overall array of possible transforms), we are able to naively separate the design of our automated pass exploration into two components:

2.2.1 *Component #1: Algorithms on Lists of Natural Numbers*

A set of transformation passes is conceptually represented as a list of natural numbers, possibly with repeats. Each number in the list maps to a specified LLVM transform pass. For example, [3,10,2,10] may correspond to the application of the LLVM transform passses [*gvn*, *simplifycfg*, *adce*, *simplifycfg*]. This representation makes enumerating the search space convenient:

```
# Pick up to max_passes elements from passes.
def uniform_selection(passes, max_passes):
return [random(0, len(passes)) for _ in range(0,
     max_passes)]

# Mutate a random pass in the list.
def mutate(passes):
    passes[random(0, len(passes))] = random(0, len(passes))

# Either mutate, add, or delete a pass in the list. This is
     a modification of a genetic/evolutionary algorithm.
def genetic(passes):
    choice = random(1, 4)
    if x==1: passes[random(0,len(passes))]=random(0,
         len(passes))
    else if x == 2: passes.append(random(0, max_passes))
    else if x == 3: passes.erase(random(0, len(passes))
    # x == 4 is no-op

# Pick a random ordering of distinct passes such that each
     pass improves the time frame significantly.
def distinct_ordering(passes):
```

Any typical search algorithms such as beam search, genetic evolution, etc. would be implementable in this framework. However, due to time constraints, our results (below) were largely obtained with simple genetic search.

2.2.2 *Component #2: A Framework for Applying an Algorithm to Select Passes*

A typical code generation DBMS engine requires significant manual effort to get the execution time results reported back into the compilation pipeline. The issue is that compile-and-optimize level statistics will happen at a different time and call site than execution-time statistics. The combination of the above profiling subsystem and our framework for pass application is ongoing and is likely to extend beyond this course; the current state of the framework is only able to gather execution-time statistics for functions which are manually invoked. To better understand the last limitation, consider the following example of a typical TPL plan for inserting into a table, which may have many functions:

```
fun Setup() { ... }
fun Body() {
    ...;
    insertIntoTable(&tableIterator, &row);
    ...;
}
fun Teardown() { ... }
```

The compiled TPL program is invoked by explicitly executing **Setup()**, **Body()**, and **Teardown()** in that order (modulo aborts or other exceptions). We can gather execution times for explicitly executed functions like **Setup()**, **Body()**, and **Teardown()**, but we cannot gather times for the **insertIntoTable()** function. Because the majority of our heuristics are currently focused on execution time, this is a significant limitation that is specific to our approach. This issue becomes even more salient when we consider the fact that

we observe disproportionate gains from optimizing these library functions, relative to what might be referred to as our "main" TPL program. See below for further discussion of results.

2.2.3 Discussion

We would like to highlight the flexibility of this two-component approach in allowing feature-engineered passes to be treated the same way as regular LLVM passes. An example is shown below.

```
{"pmenon",
[](llvm::legacy::FunctionPassManager &fpm) {
    // Add custom passes. Hand-selected based on empirical
        evaluation.
    fpm.add(llvm::createInstructionCombiningPass());
    fpm.add(llvm::createReassociatePass());
    fpm.add(llvm::createGVNPass());
    fpm.add(llvm::createCFGSimplificationPass());
    fpm.add(llvm::createAggressiveDCEPass());
    fpm.add(llvm::createCFGSimplificationPass());
}},
```

Additionally, in the DBMS context, care has to be taken to perform all profiling work in a "fake transaction", which is our phrase for a transaction that always gets aborted. This proved to be slightly tricky due to certain implementation details of how results were transmitted from our JIT'ed code back into the network layer – the code would normally write results back to the Postgres-compatible packet writer as they became available, for example. In this case, we benefit significantly from the concept of aborts being routine in a DBMS setting because it allows us to execute code that has side effects while essentially ignoring the side effects. It would be interesting to study how side effects are handled in other systems which attempt profile-guided optimization.

To track whether an improvement was truly made, we always use the minimum – noise is an issue, but noise cannot make slow code go faster, and over the many (hundreds/thousands) of iterations of our function pass applications, the minimum should win out. The algorithms that are used must ensure that mistakes can be recovered from, e.g., by allowing for arbitrary pass addition, mutation, or deletion.

3. EXPERIMENTAL SETUP

NoisePage is an end-to-end DBMS which you can connect to over SQL. Consequently, we utilize the SQL API to perform the procedure necessary to run each test.

To enable profiling, we run **SET profile_enable = true;** over SQL. This will cause queries to undergo a fixed number of profiling runs before finally executing. We recognize the following limitations of the experimental setup used for our evaluation:

- The number of profile runs and customization of profile strategies is hard-coded into our implementation for rapid iteration. There is no fundamental reason that these settings cannot be dynamically configured.
- The profiling subsystem is not yet ready to be integrated with the pass exploration framework. For this reason, profile information is neither as comprehensive nor as high-resolution as we originally intended. That said, all profile information used in our evaluation (execution times, optimize times, etc) is real, and was gathered via the TPL function instrumentation technique described above.

Specifically, we use the following procedure to perform our evaluations:

- (Optional) For a more detailed trace of what the algorithms are doing, look for and add/remove the defines `#define NOPRINT` and `#define NOPRINT2` in *llvm_optimizer.cpp* and *executable_query.cpp* respectively.
- Perform any schema setup, data loading, etc.
  - Trivial example: **CREATE TABLE foo (a int);**
  - Involved example: Load TPC-C [7], for example by using OLTPBenchmark [5]
- Enable profiling.
  - Run **SET profile_enable = true;**
- (Optional) Switch the baseline evaluation from **NOOP** (running without optimizations) to **PMENON** (expert configuration).
  - Run **SET pmenon_enable = true;**
- Execute any SQL queries. The profiling process prints to standard output.
  - Unfortunately, due to time constraints, there are no visualizers. You can CTRL-F for `PASS MARKER` to quickly jump to the baseline and/or final result.
  - The profiling data resets between SQL queries.

4. EXPERIMENTAL EVALUATION

Please see the *745_logs/* folder (provided with our code submission) for the full details of each run, including IR and passes attempted. Note that each block of `grep` output below is a separate run. The readings for `Agg original` are actually the minimum timing from at least 100 warmup runs.

4.1 SIMPLE INSERT

For an initial demonstration of our framework, we utilize an extremely simple insertion query: **INSERT INTO foo VALUES (1);**.

The first listing presents a subset of the results when run with **no optimizations** applied. The full procedure used to produce this run is as follows:

```
CREATE TABLE foo (a INT);
SET profile_enable = true;
INSERT INTO foo VALUES (1); /* 10x */
```

Selected output from this run is provided in Listing 1. Full output is provided with our code submission.

Discussion Inserting into a SQL table is such a lightweight operation that it was never profitable from an execution time standpoint to apply any optimizations to it (the `loop-unroll-and-jam` was just random noise). We will see if the expert configuration performs any better.

The next listing presents a subset of the results when run with **expert optimizations** applied. The full procedure used to produce this run is as follows:

```
CREATE TABLE foo (a INT);
SET pmenon_enable = true;
SET profile_enable = true;
INSERT INTO foo VALUES (1); /* 10x */
```

Selected output from this run is provided in Listing 2. Full output is provided with our code submission.

Discussion As stated above, it was never profitable to apply any optimizations to it – the timings for the expert picks and the search algorithm that attempted to iterate on the expert picks both did not find any significantly useful optimizations, even though the expert picks reduced the instruction count. If anything, the expert picks here were actually worse than doing anything at all: the expert picks have roughly double the optimization time, taking 0.00025 seconds more.

4.2 TPC-C PAYMENT'S SELECT CUSTOMER

We now examine a more complicated case, borrowing one of the queries executed by the TPC-C [7] PAYMENT transaction. TPC-C is a well-known database benchmark for OLTP workloads. We load with scale-factor 5.

The first listing presents a subset of the results when run with **no optimizations** applied. The full procedure used to produce this run is as follows:

```
// Load TPC-C with OLTPBenchmark --create=true --load=true
SET profile_enable = true;
SELECT C_FIRST, C_MIDDLE, C_ID, C_STREET_1, C_STREET_2,
     C_CITY, C_STATE, C_ZIP, C_PHONE, C_CREDIT,
     C_CREDIT_LIM, C_DISCOUNT, C_BALANCE, C_YTD_PAYMENT,
     C_PAYMENT_CNT, C_SINCE FROM customer WHERE C_W_ID = 2
     AND C_D_ID = 5 AND C_LAST LIKE 'a%' ORDER BY C_FIRST;
     /* 10x */
```

Selected output from this run is provided in Listing 3. Full output is provided with our code submission.

Discussion We have a similar negative result where no particular run was very profitable.

Finally, the next listing presents a subset of the results when run with **expert optimizations** applied. The full procedure used to produce this run is as follows:

```
// Load TPC-C with OLTPBenchmark --create=true --load=true
SET pmenon_enable = true;
SET profile_enable = true;
SELECT C_FIRST, C_MIDDLE, C_ID, C_STREET_1, C_STREET_2,
     C_CITY, C_STATE, C_ZIP, C_PHONE, C_CREDIT,
     C_CREDIT_LIM, C_DISCOUNT, C_BALANCE, C_YTD_PAYMENT,
     C_PAYMENT_CNT, C_SINCE FROM customer WHERE C_W_ID = 2
     AND C_D_ID = 5 AND C_LAST LIKE 'a%' ORDER BY C_FIRST;
     /* 10x */
```

Selected output from this run is provided in Listing 4. Full output is provided with our code submission.

Discussion Similarly, the expert picks take 2-3x longer to optimize at no appreciable consistent benefit to execution time. The expert picks do lower the instruction count though.

5. DISCUSSION, SURPRISES, AND LESSONS LEARNED

Discussion To summarize the 4 discussion points above (2 x **INSERT**, 2 x **SELECT**), **there was no observed appreciable benefit to applying transform passes** for OLTP workloads. Other tests that were more OLAP in nature such as **SELECT avg(i_price) FROM item**, which scans across 100k entries, also showed negligible benefit. The expert passes were quite consistent in reducing instruction count, but it did not translate to any kind of noticeable speedup. There are two possible takeaways, which is that for typical SQL queries:

- The LLVM transform passes are mostly not profitable, so don't bother with any of them.
- The LLVM transform passes are mostly not profitable, but are very cheap to run (twice as long optimizing is still fractions upon fractions of a second), so you might as well do them just in case.

A different interpretation of the results may be that the NoisePage DBMS does not expose sufficient information to the compiler to optimize its workloads. We expect that the hand-picked passes that the expert used were indeed very effective for their use-case, where data was represented as essentially raw arrays in memory and were manipulated more directly. However, in integrating their work to our NoisePage DBMS, every tuple of data is now hidden behind a function call which performs the necessary concurrency control checks, and therefore it may have been an abstraction through which the LLVM transform passes were not able to optimize.

Additionally, we found that most of the time spent optimizing a TPL program was not in the directly source-visible functions but rather in all of the supporting functions that are builtin to the TPL language. It is not clear whether optimizations are one-size fits all for all kinds of workloads, but pre-compiling these into a separate library and finding a way to link them is another possible source of optimization time savings.

One lesson that we learned from the experience in general is the distinction between a high-level design description of a solution and the development effort required to implement the solution. Near the midpoint of the project, the TUM group published the paper describing the tailored profiling technique and its applications to profile collection across abstraction levels. With the design description provided in this paper, we assumed that completing our own implementation of the technique in our profiling subsystem would prove relatively simple. This assumption proved to be incorrect. Although the core idea underlying tailored profiling, the tagging dictionary, is a simple concept, we found that populating the dictionary requires significant additions to the existing code generation and compilation infrastructure in NoisePage. The difference between the level of implementation difficulty reported in the original paper [10] and our own experience is the result of differences in execution engine architectures. While both Umbra [8] and NoisePage implement data-centric code-generation engines, Umbra's design is more amenable to the addition of support for tag collection.

6. CONCLUSION AND FUTURE WORK

In this section we present our conclusions and some potential directions for future work.

6.1 Conclusions

We identify the following technical conclusions from this project:

- We now know how to implement a profiling subsystem via tailored profiling in NoisePage. More generally, we better understand the nuances involved in lowering and preserving useful abstractions in systems with multiple levels of domain-specific intermediate representations.
- We began with the thesis of, "it may be profitable to apply different LLVM transformation passes (to different functions in the same program) in a profile-guided way to hyper-optimize a specific program".
- We conclude that, at least in the NoisePage DBMS as it integrates TPL today, you might as well not apply any optimization passes at all. You'll halve the time spent optimizing

(microseconds range) and have essentially the same execution time.

6.2 Future Work

We identify the following potential directions for future work:

- Currently, our profiling subsystem uses one of the readily available kernel time sources (Linux's `CLOCK_MONOTONIC_RAW`) to record timestamps for profile samples. In their description of tailored profiling, Beishcl et al. claim that none of the time sources exposed by the Linux kernel provide sample timestamps that are sufficiently accurate [10]. To address this issue, the authors apply a patch to the Linux kernel to expose the processor's timestamp counter register (the value read by `RDTSC` on x86) value in the profile samples written by perf. They subsequently map this timestamp counter to a wall-clock time with the help of a kernel module [10]. Future work on our profiling subsystem might include a more robust evaluation of the distribution of profile timestamps to determine if a similar approach might be necessary in our system.

- The current implementation of our profiling subsystem adopts a simple configuration in which we sample hardware instruction retirement events. However, now that we have the infrastructure in place, extending the system to sample different events and record additional data with each sample is straightforward. This presents significant opportunities for future work. For instance, Beischel et al. generate profiles for memory access patterns by sampling on memory instruction retirement events (`MEM_INST_RETIRED.ALL_LOADS`) [10]. Cache performance analysis should also be possible by sampling on the available cache reference (`PERF_COUNT_HW_CACHE_REFERENCES`) and cache miss (`PERF_COUNT_HW_CACHE_MISSES`) events.

- The current profiling framework for determining the optimal set of LLVM passes for a particular query has the ability to support many different kinds of search algorithms. Given a heuristic function that includes a combination of different profiling data points, such as memory references and execution times for different operators, it is possible to exploit the framework and implement many different search algorithms, then compare their results.

- The LLVM program that we currently optimize (which is generated in `TPL` and lowered to LLVM IR) spends a significant proportion of its overall runtime in builtin library functions, such as **tableInsert()**, **projectedRowSetInt()**, **joinHashTableInit()**, etc. Optimizing these builtin functions take a bulk of the optimization time. Therefore, it may be the case that we can pre-optimize just these functions in a one-size-fits-all manner and observe benefits similar to dynamic optimization without the associated runtime overhead.

7. WORK DISTRIBUTION

All members of our group contributed equally to this project.

8. REFERENCES

[1] hyper. `https://hyper-db.de/`.

[2] Intel 64 and ia-32 architectures software developer manuals. `https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html`.

[3] Linux perf api. `https://man7.org/linux/man-pages/man2/perf_event_open.2.html`.

[4] NoisePage. `https://noise.page`.

[5] Oltpbenchmark. `https://github.com/oltpbenchmark/oltpbench`.

[6] singlestore. `https://www.singlestore.com/db/`.

[7] Tpc-c benchmark. `http://www.tpc.org/tpcc/`.

[8] Umbra. `https://umbra-db.com/`.

[9] Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetchingwork together at last, 2018.

[10] A. Beischl, T. Kersten, M. Bandle, J. Giceva, and T. Neumann. Profiling dataflow systems on multiple abstraction levels, 2021.

[11] S. Noll, J. Teubner, N. May, and A. Bohm. Analyzing memory accesses with modern processors, 2020.

[12] B. Raducanu. Micro adaptivity in a vectorized database system, 2012.

[13] C. Stuart. Profiling compiled sql query pipelines inapache spark, 2020.

APPENDIX

Listing 1: Results of Query **INSERT INTO foo VALUES (1)** with No Optimizations

```
> grep -A 4 original: foo_insert_baseline.txt
|----| Agg original: [34 insts, 256798 opt ns, 6275 exec ns] []
|----| Agg last: [34 insts, 282244 opt ns, 6868 exec ns] [nd-correlated-value-propagation;]
|----| Agg min: [34 insts, 256798 opt ns, 6275 exec ns] []
|----| Agg mean: [31 insts, 299306 opt ns, 7228 exec ns]
|----| Agg max: [34 insts, 299812 opt ns, 28324 exec ns] [loop-simplify;]
--
|----| Agg original: [34 insts, 256828 opt ns, 6438 exec ns] []
|----| Agg last: [34 insts, 303660 opt ns, 7145 exec ns] [loop-unroll-and-jam;]
|----| Agg min: [34 insts, 261061 opt ns, 6250 exec ns] []
|----| Agg mean: [31 insts, 293810 opt ns, 7003 exec ns]
|----| Agg max: [34 insts, 296564 opt ns, 24084 exec ns] [indvars;]
--
|----| Agg original: [34 insts, 265882 opt ns, 6290 exec ns] []
|----| Agg last: [34 insts, 267068 opt ns, 6697 exec ns] [sccp;]
|----| Agg min: [34 insts, 309038 opt ns, 6199 exec ns] [loop-unroll-and-jam;]
|----| Agg mean: [31 insts, 291336 opt ns, 6963 exec ns]
|----| Agg max: [34 insts, 263768 opt ns, 12179 exec ns] [simplifycfg;]
--
|----| Agg original: [34 insts, 258198 opt ns, 6306 exec ns] []
|----| Agg last: [34 insts, 294683 opt ns, 7846 exec ns] [loop-deletion;]
|----| Agg min: [34 insts, 258198 opt ns, 6306 exec ns] []
|----| Agg mean: [31 insts, 294892 opt ns, 7073 exec ns]
|----| Agg max: [34 insts, 339107 opt ns, 20196 exec ns] [indvars;]
--
|----| Agg original: [34 insts, 266314 opt ns, 6240 exec ns] []
|----| Agg last: [34 insts, 286025 opt ns, 23367 exec ns] [loop-reduce;]
|----| Agg min: [34 insts, 266314 opt ns, 6240 exec ns] []
|----| Agg mean: [31 insts, 293613 opt ns, 7078 exec ns]
|----| Agg max: [34 insts, 286025 opt ns, 23367 exec ns] [loop-reduce;]
--
|----| Agg original: [34 insts, 258549 opt ns, 6240 exec ns] []
|----| Agg last: [34 insts, 297345 opt ns, 7028 exec ns] [indvars;]
|----| Agg min: [34 insts, 258549 opt ns, 6240 exec ns] []
|----| Agg mean: [31 insts, 308999 opt ns, 7254 exec ns]
|----| Agg max: [34 insts, 280831 opt ns, 11527 exec ns] [memcpyopt;]
--
|----| Agg original: [34 insts, 256396 opt ns, 6298 exec ns] []
|----| Agg last: [34 insts, 293989 opt ns, 6645 exec ns] [tailcallelim;]
|----| Agg min: [34 insts, 256396 opt ns, 6298 exec ns] []
|----| Agg mean: [31 insts, 296594 opt ns, 7144 exec ns]
|----| Agg max: [34 insts, 271976 opt ns, 27775 exec ns] [loop-extract-single;]
--
|----| Agg original: [34 insts, 256615 opt ns, 6146 exec ns] []
|----| Agg last: [31 insts, 349850 opt ns, 6607 exec ns] [instcombine;]
|----| Agg min: [34 insts, 256615 opt ns, 6146 exec ns] []
|----| Agg mean: [31 insts, 305433 opt ns, 7208 exec ns]
|----| Agg max: [34 insts, 272923 opt ns, 21173 exec ns] [nd-early-cse;]
--
|----| Agg original: [34 insts, 263015 opt ns, 6191 exec ns] []
|----| Agg last: [34 insts, 295891 opt ns, 6807 exec ns] [indvars;]
|----| Agg min: [34 insts, 263015 opt ns, 6191 exec ns] []
|----| Agg mean: [31 insts, 297585 opt ns, 7074 exec ns]
|----| Agg max: [34 insts, 275287 opt ns, 23212 exec ns] [loop-extract-single;]
--
|----| Agg original: [34 insts, 256223 opt ns, 6252 exec ns] []
|----| Agg last: [34 insts, 299607 opt ns, 7470 exec ns] [loop-deletion;]
|----| Agg min: [34 insts, 256223 opt ns, 6252 exec ns] []
|----| Agg mean: [31 insts, 295411 opt ns, 7015 exec ns]
|----| Agg max: [31 insts, 351603 opt ns, 9967 exec ns] [aggressive-instcombine;]
```

Listing 2: Results of Query **INSERT INTO foo VALUES (1)** with Expert Optimizations

```
> grep -A 4 original: foo_insert_expert.txt
|----| Agg original: [30 insts, 440741 opt ns, 6378 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [30 insts, 483915 opt ns, 7734 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;nd-loop-inst-simplify;]
|----| Agg min: [30 insts, 440741 opt ns, 6378 exec ns] [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg mean: [30 insts, 437485 opt ns, 7276 exec ns]
|----| Agg max: [30 insts, 470337 opt ns, 30682 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;instcombine;]
--
|----| Agg original: [30 insts, 432260 opt ns, 6524 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [30 insts, 451445 opt ns, 7381 exec ns] [aggressive-instcombine;dse;gvn;simplifycfg;adce;simplifycfg;die;]
|----| Agg min: [30 insts, 441427 opt ns, 6338 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;die;]
|----| Agg mean: [30 insts, 444231 opt ns, 7000 exec ns]
|----| Agg max: [30 insts, 472348 opt ns, 8423 exec ns] [aggressive-instcombine;reassociate;gvn;adce;adce;simplifycfg;]
--
|----| Agg original: [30 insts, 436893 opt ns, 6454 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [30 insts, 429724 opt ns, 20242 exec ns] [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;]
|----| Agg min: [30 insts, 429589 opt ns, 6412 exec ns] [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;]
|----| Agg mean: [30 insts, 434228 opt ns, 7237 exec ns]
|----| Agg max: [30 insts, 481456 opt ns, 28286 exec ns]
     [aggressive-instcombine;reassociate;gvn;loop-rotate;adce;simplifycfg;]
--
|----| Agg original: [30 insts, 436984 opt ns, 6395 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [30 insts, 437597 opt ns, 7061 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg min: [30 insts, 436984 opt ns, 6395 exec ns] [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg mean: [30 insts, 436959 opt ns, 7247 exec ns]
|----| Agg max: [30 insts, 429990 opt ns, 30359 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
--
|----| Agg original: [30 insts, 435122 opt ns, 6494 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [30 insts, 474955 opt ns, 7245 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;loop-unroll;]
|----| Agg min: [30 insts, 465651 opt ns, 6398 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;loop-unroll;]
|----| Agg mean: [30 insts, 451786 opt ns, 7045 exec ns]
|----| Agg max: [30 insts, 436202 opt ns, 11677 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
--
|----| Agg original: [30 insts, 483427 opt ns, 6613 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [30 insts, 481878 opt ns, 7298 exec ns] [aggressive-instcombine;reassociate;gvn;sccp;adce;simplifycfg;]
|----| Agg min: [30 insts, 473741 opt ns, 6593 exec ns] [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg mean: [30 insts, 449560 opt ns, 7289 exec ns]
|----| Agg max: [30 insts, 438422 opt ns, 23752 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
--
|----| Agg original: [34 insts, 276481 opt ns, 6601 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [34 insts, 268709 opt ns, 7210 exec ns] []
|----| Agg min: [34 insts, 278470 opt ns, 6425 exec ns] [sink;]
|----| Agg mean: [31 insts, 299769 opt ns, 7145 exec ns]
|----| Agg max: [34 insts, 267244 opt ns, 20372 exec ns] [sroa;]
--
|----| Agg original: [30 insts, 450926 opt ns, 6739 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [34 insts, 408464 opt ns, 6631 exec ns]
     [reassociate;gvn;simplifycfg;adce;simplifycfg;loop-unswitch;loop-rotate;]
|----| Agg min: [34 insts, 446876 opt ns, 6301 exec ns] [reassociate;gvn;simplifycfg;adce;simplifycfg;loop-unswitch;]
|----| Agg mean: [30 insts, 414105 opt ns, 7265 exec ns]
|----| Agg max: [34 insts, 420268 opt ns, 28047 exec ns]
     [dse;reassociate;gvn;simplifycfg;adce;simplifycfg;loop-unswitch;constprop;]
--
|----| Agg original: [30 insts, 432644 opt ns, 6521 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
```

```
|----| Agg last: [30 insts, 445230 opt ns, 7051 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;die;]
|----| Agg min: [30 insts, 507212 opt ns, 6465 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;die;nd-correlated-value-propagation;]
|----| Agg mean: [30 insts, 470494 opt ns, 7065 exec ns]
|----| Agg max: [30 insts, 499629 opt ns, 8474 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;die;nd-correlated-value-propagation;loop-unroll;]
--
|----| Agg original: [30 insts, 432133 opt ns, 6410 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [34 insts, 361679 opt ns, 6937 exec ns] [simplifycfg;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg min: [30 insts, 432133 opt ns, 6410 exec ns] [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg mean: [30 insts, 438346 opt ns, 7171 exec ns]
|----| Agg max: [30 insts, 420455 opt ns, 20389 exec ns] [aggressive-instcombine;gvn;simplifycfg;adce;simplifycfg;]
```

Listing 3: Results of TPC-C Query with No Optimizations

```
> grep -A 4 original: tpcc_select_baseline.txt
|----| Agg original: [127 insts, 166348 opt ns, 31350926 exec ns] []
|----| Agg last: [127 insts, 160700 opt ns, 31926567 exec ns] []
|----| Agg min: [127 insts, 202609 opt ns, 31267132 exec ns] [memcpyopt;]
|----| Agg mean: [118 insts, 231719 opt ns, 32194295 exec ns]
|----| Agg max: [127 insts, 186947 opt ns, 41145153 exec ns] [loop-extract-single;]
--
|----| Agg original: [127 insts, 163735 opt ns, 31510605 exec ns] []
|----| Agg last: [127 insts, 249558 opt ns, 32673851 exec ns] [gvn-no-load-elimination;]
|----| Agg min: [127 insts, 239359 opt ns, 31479698 exec ns] [gvn-no-load-elimination;]
|----| Agg mean: [119 insts, 238915 opt ns, 32050176 exec ns]
|----| Agg max: [127 insts, 191890 opt ns, 33436675 exec ns] [nd-early-cse;]
--
|----| Agg original: [127 insts, 165471 opt ns, 31439411 exec ns] []
|----| Agg last: [127 insts, 210252 opt ns, 32901905 exec ns] [loop-reduce;]
|----| Agg min: [127 insts, 165471 opt ns, 31439411 exec ns] []
|----| Agg mean: [118 insts, 237162 opt ns, 32481112 exec ns]
|----| Agg max: [127 insts, 234765 opt ns, 35715501 exec ns] [loop-unswitch;]
--
|----| Agg original: [127 insts, 158879 opt ns, 31426036 exec ns] []
|----| Agg last: [118 insts, 408160 opt ns, 32142559 exec ns] [instcombine;]
|----| Agg min: [127 insts, 244577 opt ns, 31272207 exec ns] [loop-unroll-and-jam;]
|----| Agg mean: [116 insts, 258419 opt ns, 31995727 exec ns]
|----| Agg max: [127 insts, 168425 opt ns, 34368579 exec ns] []
--
|----| Agg original: [127 insts, 161068 opt ns, 31267143 exec ns] []
|----| Agg last: [127 insts, 244913 opt ns, 32081382 exec ns] [gvn-no-load-elimination;]
|----| Agg min: [127 insts, 161068 opt ns, 31267143 exec ns] []
|----| Agg mean: [118 insts, 237235 opt ns, 31983379 exec ns]
|----| Agg max: [118 insts, 452599 opt ns, 33771066 exec ns] [instcombine;]
--
|----| Agg original: [127 insts, 210493 opt ns, 31680816 exec ns] []
|----| Agg last: [127 insts, 385274 opt ns, 34318157 exec ns] [dse;]
|----| Agg min: [127 insts, 449981 opt ns, 31570184 exec ns] [dse;nd-early-cse;]
|----| Agg mean: [118 insts, 363559 opt ns, 32295552 exec ns]
|----| Agg max: [127 insts, 385274 opt ns, 34318157 exec ns] [dse;]
--
|----| Agg original: [127 insts, 161069 opt ns, 31298022 exec ns] []
|----| Agg last: [127 insts, 175389 opt ns, 32197712 exec ns] [constprop;]
|----| Agg min: [127 insts, 161069 opt ns, 31298022 exec ns] []
|----| Agg mean: [118 insts, 230877 opt ns, 31988596 exec ns]
|----| Agg max: [127 insts, 194860 opt ns, 37799014 exec ns] [jump-threading;]
--
|----| Agg original: [127 insts, 163623 opt ns, 31711482 exec ns] []
|----| Agg last: [127 insts, 262709 opt ns, 33503415 exec ns] []
|----| Agg min: [127 insts, 193378 opt ns, 31179025 exec ns] [adce;]
|----| Agg mean: [118 insts, 225709 opt ns, 32390057 exec ns]
|----| Agg max: [127 insts, 170274 opt ns, 36173310 exec ns] []
--
|----| Agg original: [127 insts, 183864 opt ns, 31623651 exec ns] []
|----| Agg last: [127 insts, 267010 opt ns, 32960109 exec ns] [indvars;reassociate;]
|----| Agg min: [127 insts, 198099 opt ns, 31155732 exec ns] [die;reassociate;]
|----| Agg mean: [118 insts, 223547 opt ns, 32106866 exec ns]
|----| Agg max: [127 insts, 178237 opt ns, 34549015 exec ns] [die;]
--
|----| Agg original: [127 insts, 165136 opt ns, 31526410 exec ns] []
|----| Agg last: [127 insts, 190163 opt ns, 32252023 exec ns] [nd-early-cse;]
|----| Agg min: [127 insts, 393706 opt ns, 31414729 exec ns] [dse;nd-early-cse;]
|----| Agg mean: [122 insts, 321612 opt ns, 32146519 exec ns]
|----| Agg max: [127 insts, 373848 opt ns, 33946677 exec ns] [dse;]
```

Listing 4: Results of TPC-C Query with Expert Optimizations

```
> grep -A 4 original: tpcc_select_expert.txt
|----| Agg original: [113 insts, 599769 opt ns, 31279871 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [113 insts, 592277 opt ns, 32964854 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg min: [113 insts, 599769 opt ns, 31279871 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg mean: [113 insts, 632874 opt ns, 33061751 exec ns]
|----| Agg max: [113 insts, 937190 opt ns, 39582158 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
--
|----| Agg original: [113 insts, 604514 opt ns, 31736279 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [127 insts, 532395 opt ns, 32536341 exec ns] [dse;gvn;adce;simplifycfg;]
|----| Agg min: [127 insts, 376043 opt ns, 31683757 exec ns] [reassociate;gvn;adce;simplifycfg;]
|----| Agg mean: [113 insts, 418862 opt ns, 32617899 exec ns]
|----| Agg max: [113 insts, 589678 opt ns, 37875476 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
--
|----| Agg original: [113 insts, 734860 opt ns, 31325576 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [113 insts, 804062 opt ns, 32226713 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;dse;]
|----| Agg min: [113 insts, 718675 opt ns, 31289834 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;aggressive-instcombine;]
|----| Agg mean: [113 insts, 689353 opt ns, 32406489 exec ns]
|----| Agg max: [113 insts, 799925 opt ns, 37202479 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;licm;aggressive-instcombine;]
--
|----| Agg original: [113 insts, 606854 opt ns, 31427134 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [127 insts, 573497 opt ns, 49375678 exec ns] [reassociate;gvn;simplifycfg;adce;sroa;memcpyopt;sink;sroa;gvn;]
|----| Agg min: [127 insts, 416310 opt ns, 30791729 exec ns] [reassociate;gvn;simplifycfg;adce;sroa;memcpyopt;sink;sroa;]
|----| Agg mean: [113 insts, 574880 opt ns, 31831876 exec ns]
|----| Agg max: [127 insts, 573497 opt ns, 49375678 exec ns] [reassociate;gvn;simplifycfg;adce;sroa;memcpyopt;sink;sroa;gvn;]
--
|----| Agg original: [113 insts, 653554 opt ns, 31260410 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [113 insts, 663885 opt ns, 34415877 exec ns]
     [aggressive-instcombine;loop-deletion;gvn;simplifycfg;simplifycfg;]
|----| Agg min: [113 insts, 624586 opt ns, 31007677 exec ns]
     [aggressive-instcombine;loop-deletion;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg mean: [113 insts, 633591 opt ns, 32101015 exec ns]
|----| Agg max: [113 insts, 618316 opt ns, 37792914 exec ns]
     [aggressive-instcombine;loop-deletion;gvn;simplifycfg;adce;simplifycfg;]
--
|----| Agg original: [113 insts, 592692 opt ns, 31018692 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [113 insts, 677768 opt ns, 32705825 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;loop-reduce;]
|----| Agg min: [113 insts, 592692 opt ns, 31018692 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg mean: [113 insts, 602194 opt ns, 32147835 exec ns]
|----| Agg max: [113 insts, 671044 opt ns, 43641554 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;jump-threading;]
--
|----| Agg original: [113 insts, 733086 opt ns, 31109160 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [113 insts, 606790 opt ns, 32806305 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg min: [113 insts, 733086 opt ns, 31109160 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg mean: [113 insts, 608434 opt ns, 32155034 exec ns]
|----| Agg max: [113 insts, 602517 opt ns, 38787336 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
--
|----| Agg original: [113 insts, 662951 opt ns, 31319957 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [113 insts, 598255 opt ns, 32026952 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg min: [113 insts, 662951 opt ns, 31319957 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
```

```
|----| Agg mean: [113 insts, 614480 opt ns, 32078331 exec ns]
|----| Agg max: [127 insts, 426075 opt ns, 34824591 exec ns]
     [loop-unroll-and-jam;reassociate;gvn;simplifycfg;adce;simplifycfg;]
--
|----| Agg original: [113 insts, 590692 opt ns, 31275904 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [127 insts, 590911 opt ns, 31779946 exec ns]
     [nd-early-cse;reassociate;gvn;simplifycfg;adce;nd-gvn;loop-unroll;]
|----| Agg min: [127 insts, 610072 opt ns, 30903603 exec ns]
     [nd-early-cse;reassociate;gvn;simplifycfg;adce;nd-gvn;loop-unroll;]
|----| Agg mean: [113 insts, 629285 opt ns, 31931582 exec ns]
|----| Agg max: [113 insts, 854839 opt ns, 35007018 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;nd-gvn;loop-unswitch;]
--
|----| Agg original: [113 insts, 616703 opt ns, 31184050 exec ns]
     [aggressive-instcombine;reassociate;gvn;simplifycfg;adce;simplifycfg;]
|----| Agg last: [118 insts, 499051 opt ns, 32786169 exec ns] [aggressive-instcombine;reassociate;sccp;adce;simplifycfg;]
|----| Agg min: [118 insts, 547734 opt ns, 31002474 exec ns]
     [aggressive-instcombine;reassociate;sccp;simplifycfg;adce;simplifycfg;]
|----| Agg mean: [113 insts, 588220 opt ns, 31849649 exec ns]
|----| Agg max: [113 insts, 592411 opt ns, 36662351 exec ns] [aggressive-instcombine;reassociate;gvn;adce;simplifycfg;]
```